Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Žiga Putrle

Nadgradnja tolmača konkatenacijskega programskega jezika za spremljanje izvajanja programa

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Jurij Mihelič

Ljubljana, 2017

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Ugotavljanje ustavljivosti programa je v splošnem neizračunljiv problem, kljub temu pa v praksi za marsikateri program znamo povedati ali se bo ustavil ali ne. Prav tako lahko z opazovanjem samega programa tekom izvajanja marsikdaj napovemo njegovo obnašanje. Za takšno opazovanje potrebujemo posebno izvajalno okolje, ki omogoča beleženje stanj izvajajočega se programa. V okviru diplomske naloge se osredotočite na enega izmed konkatenacjskih programskih jezikov. Predelajte enega izmed obstoječih tolmačev za izbrani jezik, da bo omogočal beleženje in opazovanje stanj programa. Predelavo ovrednotite s preprostim testom ugotavljanja ustavljivosti preko ponavljanja stanja programa.

Zahvaljujem se mentorju, doc. dr. Juriju Miheliču za strokovno pomoč pri izdelavi diplomskega dela. Zahvaljujem se staršem, da so mi omogočili izobraževanje. Zahvaljujem se družini za spodbudo in potrpljenje, ki so mi ju izkazali. In nenazadnje, zahvaljujem se prijateljem, ki so poskrbeli, da je bilo študijsko življenje zabavno in polno.

Družini in prijateljem

Kazalo

Povzetek

Naslov: Nadgradnja tolmača konkatenacijskega programskeg jezika za spremljanje izvajanja programa

Diplomsko delo opisuje načrtovanje, nadgradnjo in preizkus tolmača, ki omogoča spremljanje izvajajočega programa, zapisanega v konkatenacijskem programskem jeziku. Uporabili smo programski jezik Forth, preučili njegovo delovanje, izbrali tolmač, ga nadgradili, izdelali potrebno programsko opremo in izvedli elementarne teste, ki so potrdili primernost opreme. Namen izdelane programske opreme je ponuditi ustrezno okolje za opazovanje in analizo neizračunljivih problemov.

Ključne besede: Forth, tolmač, navidezni stroj, zbiranje podatkov, programski jezik C.

Abstract

Title: Upgrade of a concatenative programming language interpreter for monitoring of executing program

The graduation thesis describes the design, upgrade and testing of computer software that supports monitoring of a running program written in concatenative programing language. We have used the Forth programming language, studied its design, selected the interpreter, upgrade it, create the support software, and conducted the elemental tests that have confirmed that the created software is appropriate. The purpose of the software is to provide an environment for the observation and analysis of decision problems.

Keywords: Forth, interpreter, virtual machine, data collection, C programming language.

Poglavje 1

Uvod

Ali se bo program ustavil? To je vprašanje, ki je porodilo temo diplomskega dela. Problem ustavljivosti in preostali neizračunljivi problemi so teoretično nerešljivi. Zanima nas, kako se njihovo reševanje obnese v praksi. Menimo, da z opazovanjem in analizo izvajajočega programa lahko v nekaterih primerih napovemo, ali se bo program ustavil.

Namen diplomskega dela je izdelava programske opreme, ki omogoča opazovanje izvajajočega programa in nudi podporo nadaljnji analizi. Kot jezik, uporabljen za analizo, smo izbrali programski jezik Forth. Preučili smo njegovo delovanje, izbrali tolmač in ga nadgradili z interaktivnim vmesnikom, ki omogoča zbiranje podatkov o izvajajočem programu. Izdelali smo potrebno programsko opremo in izvedli elementarne teste, ki so potrdili delovanje nadgradnje.

Prvi del diplomskega dela vsebuje teoretični uvod v svet Forth programskega jezika. Obravnavajo se osnove Fortha, delo navideznega stroja, sestavo slovarjev in delovanje notranjega ter zunanjega tolmača. V drugem delu se teoretični del razširi z opisom nadgradnje. Pojasnjene so razvojne odločitve, predstavljena je nadgradnja in opisan je postopek testiranja.

Poglavje 2

Forth programski jezik

Za razumevanje nadgradnje je pomembno osnovno razumevanje programskega jezika Forth in delovanje njegovega tolmača. Poglavje vsebuje kratko predstavitev programskega jezika Forth.

2.1 Zbiranje podatkov

Zbiranje podatkov je proces sistematičnega pridobivanja in merjenja informacij o želenem pojavu. Pojav, ki ga bomo opazovali, je program, zapisan v Turing polnem (ang. *Turing complete*) programskem jeziku. Lastnosti izbranega programskega jezika vplivajo na proces zbiranja podatkov, analize in izdelavo programske rešitve. Pri izbiri jezika smo upoštevali naslednja vodila. Programski jezik mora podpirati konkatenacijsko paradigmo, biti Turing popoln in enostaven. Skupina teh lastnosti nam omogoča enostavnejšo analizo in zbiranje podatkov ter podpira izdelavo enostavnih in kompleksnih programov. Izbrali smo programski jezik Forth.

Forth je tolmačen programski jezik. Uporaba tolmača doprinese dodatno abstrakcijo med nosilnim sistemom in izvajajočim programom. Delovanje tolmača smo uporabili za definicijo podatkovnega modela in izdelavo programske rešitve.

Vprašanje primernosti podatkov in njihovih informacij ni del diplomskega

dela. Rešitev zato vsebuje dinamični vmesnik, ki omogoča široko izbiro podatkov in enostavno nadgradnjo. Zaradi eksperimentalne narave izvora diplomske naloge je pomembno, da omogočimo in ohranimo razširljivost nadgradnje. Kasnejše dodajanje zahtev bo zato mogoče in enostavnejše. Želimo si, da bo končana programska oprema primerna tudi za analizo širše domene odločitvenih problemov (ang. *decision problem*).

2.2 Konkanetacijska paradigma

Konkatenacijska paradigma je skupina lastnosti, s katerimi lahko opišemo programski jezik [5, 6]. Ime konkatenacijski jezik izvira iz teorije jezikov, pri čemer se konkatenacija nanaša na sestavljanje nizov črk. Vzorec se uporablja pri izražanju dejanj v programski kodi. Z nizanjem in medsebojno kompozicijo funkcij izrazimo želeno dejanje. Primer 2.1 prikazuje, kako z nizom funkcij predstavimo operacijo zapiranja ventila. Vsako neprekinjeno zaporedje znakov predstavlja funkcijo.

```
1 IF VALVE_OPEN THEN VALVE_CLOSE ELSE ECHO_OK ENDIF
```

Primer 2.1: Primer kompozicije

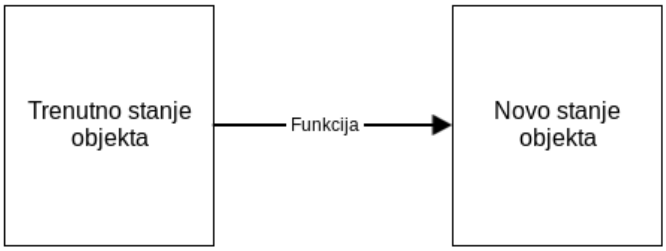
Osrednji objekt predstavlja rdečo nit konkatenacijskega jezika in je ponavadi realiziran v obliki sklada. Uporablja se za prenos argumentov in izvajanje opracij. Posledice izvedenih funkcij se odražajo na njem in preko njega lahko sledimo stanju programa. Slika 2.1 prikazuje spremembo stanja objekta.

Argumentov ne navajamo eksplicitno, saj jih funkcije pridobijo preko osrednjega objekta. To uvršča konkatenacijske jezike v skupino *point-free* programskih jezikov [6].

```
1 10 20 +
```

Primer 2.2: Primer seštevanja

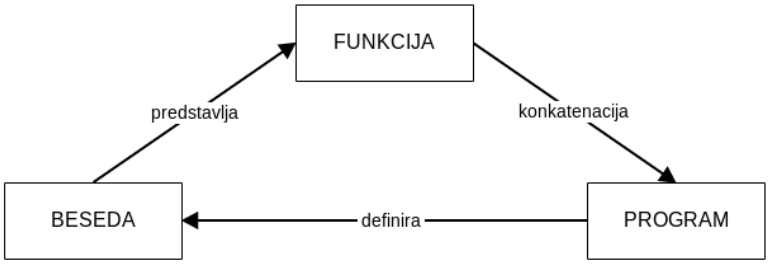
Tok operacij v primeru 2.2 teče iz leve proti desni. Uporabljena je obrnjena poljska notacija. Števili 10 in 20 se ne obravnavata kot števili, ampak



Slika 2.1: Sprememba stanja osrednjega objekta

funkcije, katerih učinek je dodano število na osrednji objekt. Beseda + odvzame števili, ju sešteje in rezultat vrne na objekt. Uporaba besede 'funkcija' in 'beseda' se v kontekstu konkatenacijskih jezikov uporablja izmenično.

Sintaksa konkatenacijskih jezikov je preprosta, saj obravnava samo funkcije in njihovo medsebojno kompozicijo. Poenostavitev se izraža s preprostejšim tokom operacij, kar omogoča enostavnejšo implementacijo. Optimizacijo programske kode lahko dosežemo s preoblikovanjem programa (ang. *refactoring*) ali uporabo paralelizma, kjer se izvedejo neodvisne besede hkrati. Preprostost konkatenacijskih jezikov se odraža pri definiciji novih besed. Prikazano na sliki 2.2. Beseda v jeziku predstavlja funkcijo. S kompozicijo funkciji lahko izdelamo program. Program lahko definira novo besedo.



Slika 2.2: Definicija besede

Forth ima enostavno sintakso, kar nam skupaj s centralnimi objekti omogoča enostavno predstavitev sistema. Slabost konkatenacijskih jezikov se izrazi pri preglednosti programske kode. Uporaba centralnega objekta spremeni

tok podatkov, ki mu je posledično težje slediti. Potreben je dodaten trud za vzdrževanje berljivosti programske kode. To lahko dosežemo z jasno definicijo funkcij in njihovega učinka na objekt.

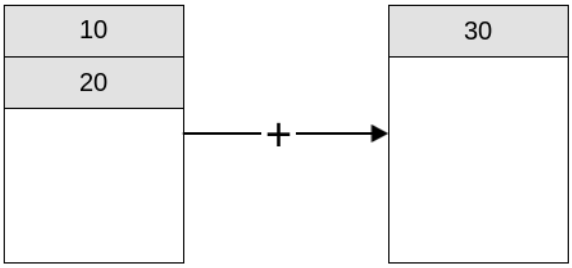
2.3 Forth

Začetki programskega jezika Forth segajo v leto 1968, kjer ga je Charles H. Moore razvijal in uporabljal kot osebni programski jezik [7]. S pomočjo Elizabeth Rather sta Forth leta 1973 razširila na desetine različnih sistemov. Jezik je postal popularen v osemdesetih letih, saj se je zaradi svoje velikosti in prenosljivosti prilegal mnogim sistemom z omejenimi sredstvi. Standardizacija jezika je potekala v letih 1979 in 1983. Zadnjo različico standarda je izdal ANSI leta 1994 [15].

Ime FORTH izvira iz leta 1968. Datoteka, ki je vsebovala tolmač, je bila poimenovana FOURTH, kot četrta generacija tolmača. Vendar je operacijski sistem, prisoten na IBM 1130, podpiral le imena datotek s štirimi znaki. Zato so ga preimenovali v FORTH. Beseda FORTH ni kratica. Zapis z velikimi črkami se je ohranil iz zgodnjih let uporabe [7]. Danes se uporablja zapis FORTH in Forth.

Forth podpira imperativno programiranje (ang. *imperative programming*), refleksijo (ang. *reflection*) in konkatenacijsko programiranje (ang. *concatenation programming*). Je preprost in majhen, vendar omogoča široko razširljivost. Implementacija jezika temelji na skladu (ang. *stack*). Jezik ne vsebuje preverjanja tipov. Implementacije pogosto vsebujejo interaktivno lupino, ki omogoča lažji pregled in razhroščevanje. Razširljivost jezika omogoča uporabo visoko nivojskih operacij.

Osnovna enota jezika je beseda, ki je definirana z vnosom v slovar. Slovar je skupek definicij, ki predstavlja jedro jezika. Del slovarja je predhodno definiran in je na voljo, ko se program prenese v polnilnik. Večino besed lahko opišemo z njihovim učinkom na sklad. Slika 2.3 prikazuje spremembo sklada ob izvedbi besede +.



Slika 2.3: Izvedba besede +, glej primer 2.2.

Besede so predstavljene kot zaporedje vidnih znakov, kjer so beli znaki uporabljeni kot mejniki (ang. *delimiters*). Primer 2.3 prikazuje primere veljavnih bese 'ventil', 'DROP', 'X', '123', ':' in '//II'.

```
1  ventil DROP X 123 : //II
```

Primer 2.3: Veljavne besede

Besede delimo na osnovne in sestavljene. Osnovne besede vsebuje strojno kodo nosilnega sistema in so jedro slovarja. Predstavljajo najnižjo enoto jezika in so uporabljene za definicijo preostalih besed, ki se imenujejo sestavljene besede. Uporabljamo jih za definicjo kompleksnejših operacij. [10, 16]

Dandanes se Forth uporablja za programiranje sistemov z omejenimi sredstvi. Primer uporabe sta vesoljsko plovilo Philae [1] in del zagonskega sistema FreeBSD [2].

2.4 Osnovne besede

Besede lahko glede na njihov učinek razdelimo v različne skupine. V tem poglavju so opisane nekatere izmed teh skupin, ki so potrebne za osnovno razumevanje Forth programskega jezika.

2.4.1 Delo s skladom

Besede, uporabljene pri delu s skladom.

```
DROP    (x - )
  Odstrani vrhnjo vrednost na skladu.
```

```
DUP    (x - x x)
  Podvoji vrhnjo vrednost na skladu.
```

```
SWAP    (x1 x2 - x2 x1)
  Zamenja prvi dve vrednosti na skladu.
```

```
ROT    (x1 x2 x3 - x2 x3 x1)
  Prestavi prve tri vrednosti na skladu.
```

2.4.2 Pomoč programerju

Besede, namenjene v pomoč programerju.

```
.S    ( - )
   Prikaže vsebino sklada. Vsebina sklada se ne spremeni.
```

```
?    ( a-addr - )
  Prenos in prikaz vrednosti na podanem naslovu.
```

```
.    ( x - )
  Izpis vrhnje vrednosti na skladu.
```

2.4.3 Aritmetične in logične operacije

Besede, namenjene aritmetičnim in logičnim operacijam. Uporablja se obrnjena poljska notacija.

```
+    ( n1 n2 - n3 )
  Seštej prvi dve vrednosti na skladu.
```

```
-    ( n1 n2 - n3 )
  Odštej n2 od n1.
```

```
*    ( n1 n2 - n3 )
  Množenje prvih dveh vrednosti na skladu.
```

```
\     ( n1 n2 – n3 )
  Deli n1 z n2.
```

```
LSHIFT     ( x1 u – x2 )
  Logični levi premik vrednost x1 za u mest. Na desni strani so
   dodajo ničle.
```

```
AND     ( x1 x2 – x3 )
  IN logična operacija med soležnimi biti.
```

```
OR     ( x1 x2 – x3 )
  ALI logična operacija med soležnimi biti.
```

2.4.4 Delo s pomnilnikom

Besede, namenjene delu s pomnilnikom.

```
!     ( x a–addr – )
  Shrani vrednost x na naslov a–addr.
```

```
@     ( a–addr – x)
  Prenese vrednost x iz naslova a–addr na sklad.
```

```
VARIABLE <ime>     ( – )
  Definicija spremenljivke. Naredi vnos v slovarju velikosti
  ene celice z imenom <ime>.

  Primer uporabe:

  VARIABLE LETO
  2016 LETO ! \ shranimo vrednost 2016 v spremenljivko LETO
  LETO @ \ prenesemo vrednost (2016) iz spremenljivke na sklad

  Kadar uporabimo ime spremenljivke LETO, se na sklad prenese
   naslov, kjer se njena vrednost nahaja.
```

2.4.5 Nadzor toka besed

Besede, namenjene pogojni izvedbi besed.

```
=    ( n1 n2 - flag )
  Zapiše vrednost TRUE na sklad, če sta vrednosti n1 in n2
  enaki. Sicer zapiše FALSE.
```

```
<    ( n1 n2 - flag )
  Zapiše vrednost TRUE na sklad, če je vrednost n1 manša od
  vrednosti n2. Sicer zapiše FALSE.
```

```
>    ( n1 n2 - flag )
  Zapiše vrednost TRUE na sklad, če je vrednost n1 večja od
  vrednosti n2. Sicer zapiše FALSE.
```

IF-ELSE stavek:

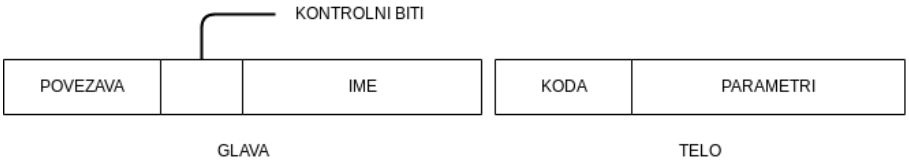
```
<testna vrednost> IF
    <če TRUE besede>
ELSE
    <če FALSE besede>
THEN

Primer uporabe:

LETO @ DUP 2017 = IF
    ." Leto je 2017" CR
ELSE
    ." Leto je " . CR
THEN
```

2.5 Slovar

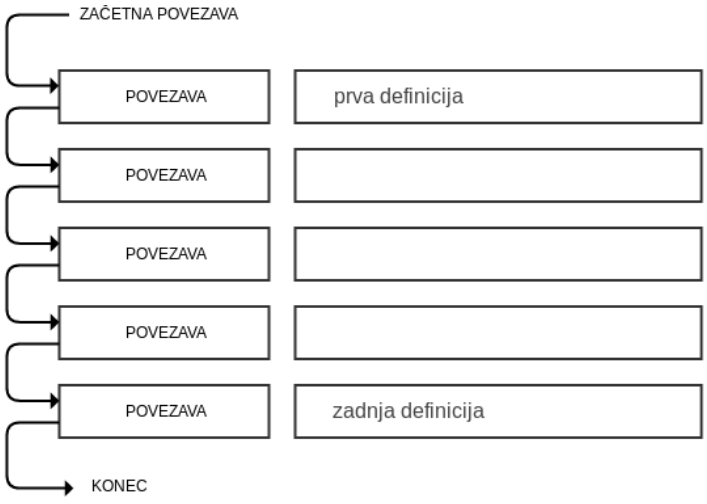
Slovar je objekt, ki vsebuje definicije besed [11]. Osnovna skupina je predhodno definirana in je na voljo ob zagonu programa. Vsaka beseda ima svojo definicijo, ki opisuje njeno delovanje in je uporabljena v času tolmačenja. Slika 2.4 predstavlja osnovno definicijo besede v slovarju [17].



Slika 2.4: Definicija besede [11]

Definicijo lahko razdelimo na glavo in telo. Glava vsebuje identiteto besede, telo pa je nosilec rutine, ki se izvede ob njenem klicu. Osnovni deli definicije so;

- <POVEZAVA>, ki kaže na naslednjo definicijo,
- <KONTROLNI BITI>, ki prilagajajo obnašanje besede,
- <IME>, ki je znakovna predstavitev besede,
- <KODA>, ki je skupina besed ali strojna koda, izvedena ob klicu in
- <PARAMETRI>, ki so podatki, uporabljeni ob izvedbi besede.



Slika 2.5: Zgradba slovarja [11]

Slovar je organiziran v povezani seznam (ang. *linked list*). Slika 2.5 prikazuje zgradbo slovarja. Z uporabo povezanega seznama je omogočeno dinamično dodajanje besed in njihova redefinicija. Novejše besede se nahajajo bližje začetni povezavi. Za definicijo nove besede se uporablja naslednja sintaksa.

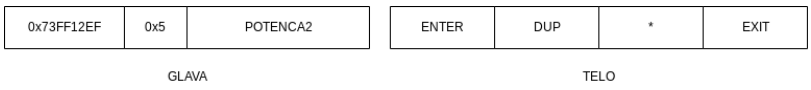
```
: <IME> <JEDRO> ;
```

Beseda ':' označuje začetek definicije nove besede z imenom <IME>. Sledi <JEDRO>, sestavljeno iz skupine besed, ki predstavljajo obnašanje nove besede. Definicijo zaključimo z besedo ';'. Primer 2.4 prikazuje definicijo nove besede.

```
1 : POTENCA2 DUP * ;
```

Primer 2.4: Definicija besede POTENCA2

Beseda POTENCA2 predstavlja drugo potenco vrhnje vrednosti sklada. Slika 2.6 prikazuje simbolični prikaz zapisa definicije v slovarju [11]. Besedi ENTER in EXIT sta pojasnjeni kasneje, glej poglavje 2.8.



Slika 2.6: Simboličen prikaz definicije besede POTENCA2

2.6 Navidezni stroj

Jedro Forth tolmača sestavlja navidezni stroj, katerega naloga je izvajanje ukazov. Sestavljen je iz dveh skladov, logične enote in nekaj registrov [11]. Lastnosti navideznega stroja se lahko razlikujejo med posameznimi implementacijami, saj so te močno vezane na nosilni sistem.

Slika 2.7 predstavlja gradnike Forth navideznega stroja. Uporabljajo se tri ali dva pomensko ločena sklada. Podatkovni sklad (ang. *Data Stack*) vsebuje splošne podatke in ga uporablja večina besed. Zaradi pogoste uporabe

ga imenujejo tudi 'The Stack'. Vrnitveni sklad (ang. *Return Stack*) nosi rezultate izvedenih besed, kontrolni sklad (ang. *Control Stack*) pa se uporablja za hranjenje podatkov, potrebnih za nadzor besednega toka. Običajno sta vrnitveni in kontrolni sklad združena, saj je njuna uporaba redka. Skladi so ključnega pomena za prenos informacij v navideznem stroju. Implementacije, ki podpirajo paralelno izvajanje več procesov, imajo lahko več skladov z enako vlogo [11].

Slika 2.7: Forth navidezni stroj

Za podporo logične enote so pogosto uporabljeni naslednji registri:

- **SP** (ang. *Data Stack Pointer*) kaže na vrh podatkovnega sklada.
- **RP** (ang. *Return Stack Pointer*) kaže na vrh vrnitvenega sklada.
- **IP** (ang. *Instruction Pointer*) kaže na naslednjo besedo, ki bo izvedena.
- **WP** (ang. *Word Pointer*) kaže na trenutno definicijo besede, ki se izvaja.

Dodani so lahko še registri, kot je **UP** (ang. *User pointer*), ki pri večprocesnih programih kaže na trenutno izvajajočo rutino.

Logična enota (LE) je zadolžena za izvajanje besed in sledenje besednemu toku. Njeno jedro predstavlja notranji tolmač, glej poglavje 2.8.

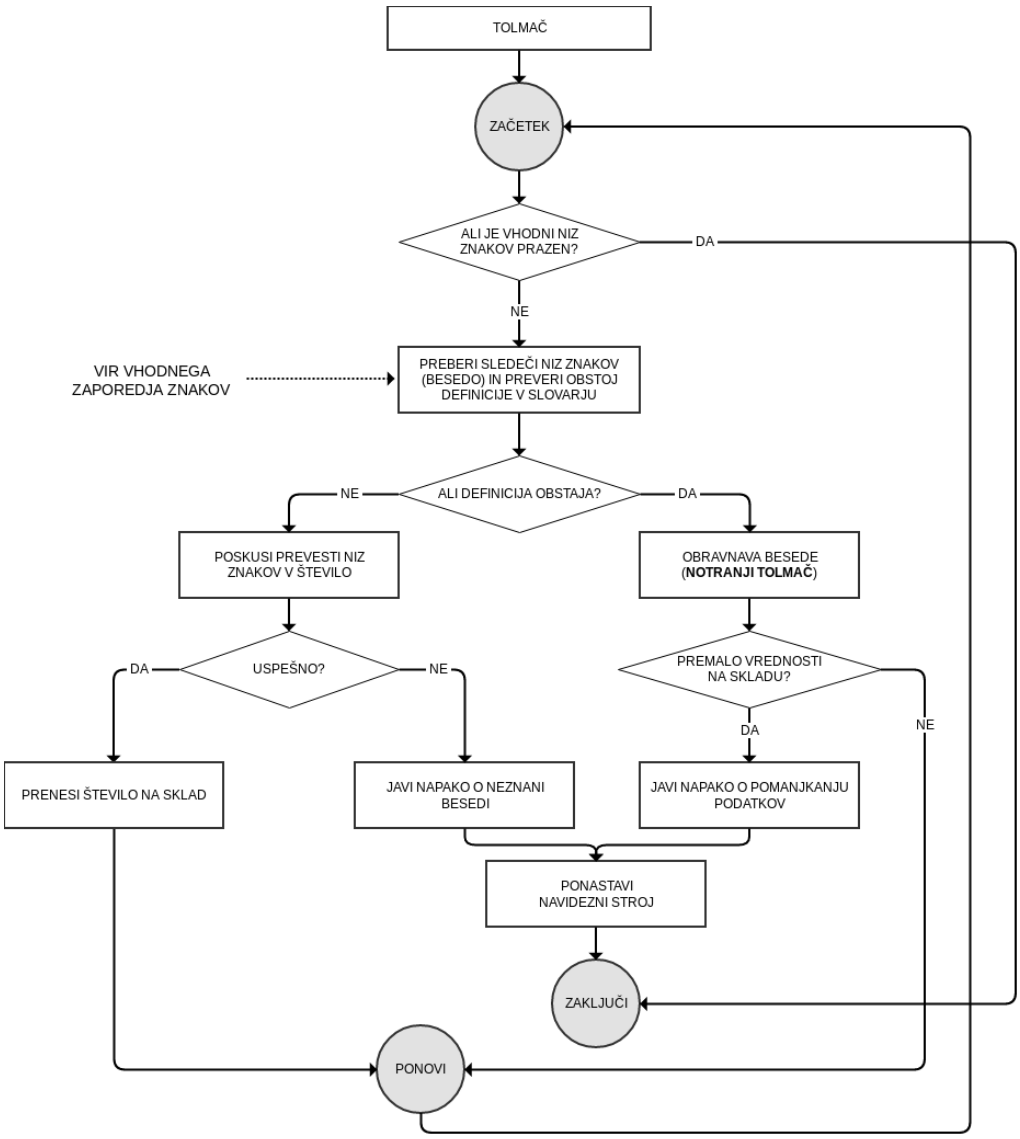
2.7 Zunanji tolmač

Zunanji tolmač je del logične enote, ki skrbi za obravnavo vhodnega zaporedja znakov. Vir vhodnega zaporedja je datoteka ali pa interaktivna lupina. Delovanje tolmača je prikazano na sliki 2.8 [12, 14, 17].

Zunanji tolmač ponavlja naslednje korake do izteka vhodnega zaporedja znakov ali do prve napake.

1. Preberi neprekinjeno zaporedje vidnih znakov iz vhodnega zaporedja.
2. Poišči definicijo besede v slovarju z enako znakovno predstavitvijo.
3. Če primerna definicija obstaja, obravnavaj besedo. Če primerne definicije ni, niz pretvori v število in ga dodaj na sklad.
4. Nadaljuj na prvem koraku. V primeru napake ustavi delovanje in sporoči napako.

Obravnavanje besed se odvija v notranjem tolmaču, glej poglavje. 2.8



Slika 2.8: Zunanji tolmač [12]

2.8 Notranji tolmač

Notranji tolmač skrbi za obravnavo besed in je del zunanjega tolmača. Sestavlja ga zaporedje besed NEXT, ENTER in EXIT. Skupaj predstavljajo rutino, ki omogoča zaporedno izvajanje besed. Prikaz naslednjih operacij je simboličen, saj je delovanje notranjega tolmača močno vezano na nosilni

sistem. Definicija besed:

```
NEXT  ( izvedi besedo )
    WP = IP
    IP = naslednja beseda
    IZVEDI WP
```

```
ENTER  ( vstopi v sestavljeno besedo )
    PUSH IP / register IP shranimo na vrnitveni sklad
    IP = prva beseda znotraj sestavljene besede
```

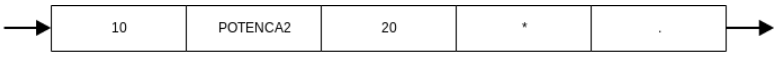
```
EXIT ( izstopi iz sestavljene besede )
    POP IP / povrnemo vrednost s sklada v register IP
```

Primer izračuna prostornine kvadra prikazuje delovanje notranjega tolmača. Predpostavljeno je, da je višina in širina kvadra enaka 10 cm in da je dolžina enaka 20 cm. Primer 2.5 prikazuje izračun v Forth programskem jeziku.

```
1  10 POTENCA2 20 * .
```

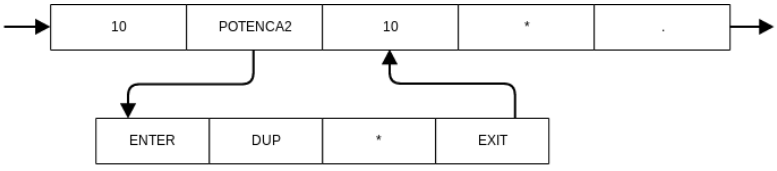
Primer 2.5: Izračun prostornine kvadra

Program nam pove, da je ploščina prizme enaka 2000 cm^3. Beseda '.' izpiše vrhnjo vrednost sklada. Slika 2.9 prikazuje simbolični prikaz programa.



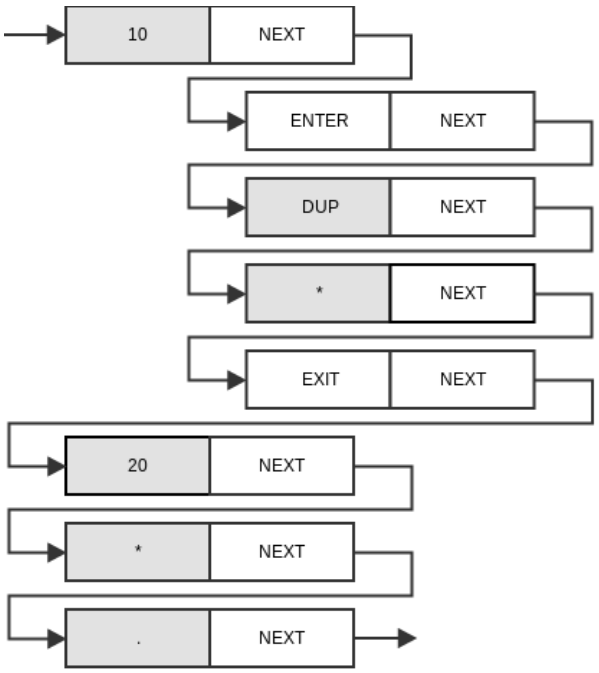
Slika 2.9: Prikaz toka besed, primera 2.5.

Besede se izvajajo od leve proti desni. Če predpostavimo, da je POTENCA2 edina sestavljena beseda, lahko prikaz razširimo, glej sliko 2.10.



Slika 2.10: Prikaz toka besed, primer 2.5, z razširitvijo besede POTENCA2.

Puščice prikazujejo preusmeritev besednega toka. Ker je POTENCA2 sestavljena beseda, tolmač izvede besede, ki so zapisane v njeni definiciji. Dodamo še besede notranjega tolmača NEXT, ENTER in EXIT ter dobimo končno zaporedje izvedenih besed.



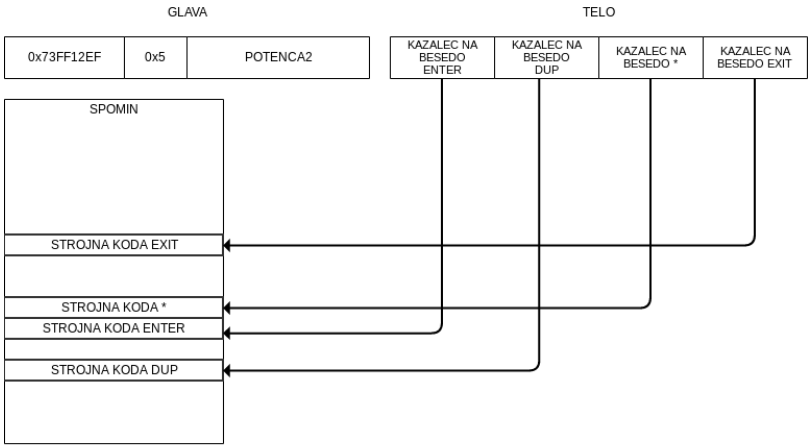
Slika 2.11: Razširjeno zaporedje ukazov primera 2.5.

Iz primera 2.11 je razvidno, da se beseda NEXT izvede po vsaki besedi razen sama po sebi. Besedi ENTER in EXIT pa se uporabljata za vhodni in izhodni del sestavljenih besed. Beseda ENTER predstavlja prolog in beseda EXIT epilog sestavljene besede.

Delovanje notranjega tolmača je vezano na obliko predstavitve besed, saj to določi vsebino definicije in njeno obravnavo. Uporablja se naslednje metode.

Posredno-nitena koda (ang. *Indirect-threaded code*) se s kazalci sklicuje na lokacijo v pomnilniku, kjer se nahaja rutina [13, 18]. Kazalce se ob definiciji shrani v telo besede in se jih kasneje uporabi pri njeni izvedbi.

Beseda NEXT uporabi ukaz JUMP za preusmeritev besednega toka. Proces je primerljiv z definicijo podrutine. Na sliki 2.12 je prikazano delovanje posredno-nitene kode.



Slika 2.12: Simboličen primer posredno-nitene koda

Neposredno-nitena koda (ang. *Direct-threaded code*) ob definiciji nove besede prenese kodo v telo definicije. S tem se izogne uporabi ukazov JUMP [18].

Podrutinsko-nitena koda (ang. *Subroutine-threaded code*) doda vsaki besedi v definicijo ukaz JUMP. Učinek je enakovreden notranjemu tolmaču. Posledično notranji tolmač ni več potreben.

Žetonsko-nitenje kode (ang. *Token-threaded code*) predstavi besede z edinstvenimi žetoni (ang. *token*). Žetoni se med tolmačenjem preslikajo v kazalec na spomin, kjer se rutina nahaja [13].

Poglavje 3

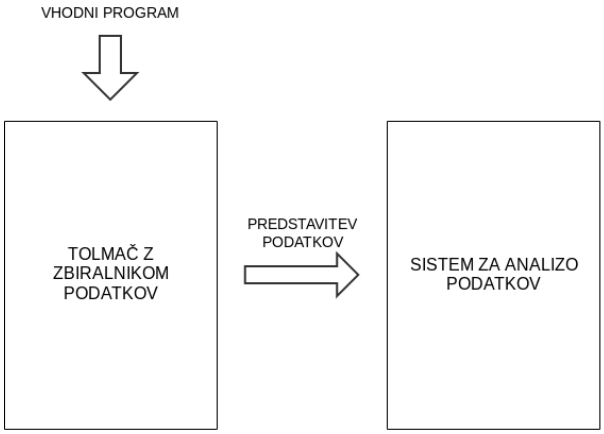
Nadgradnja

V začetnih korakih načrtovanja smo imeli željo po implementaciji tolmača, ki bi bil v celoti prilagojen našim potrebam. Zaradi kompleksnosti implementacije se je kmalu izkazalo, da bi bila količina vloženega časa prevelika in nepotrebna. Z obstojem modernih odprto licenčnih implementacij (GForth in PForth) bi večino našega dela spadalo pod implementacijo že obstoječe programske opreme. Odločili smo se, da lahko z nadgradnjo že obstoječe implementacije dosežemo enakovredno v krajšem času. Za nadaljnjo analizo delovanja obstoječih tolmačev smo kot najprimernejšo izbrali implementacijo PForth. Nadgradnja obsega implementacijo vmesnika, ki omogoča zbiranje podatkov o delovanju tolmača. Programi lahko preko njega dostopajo do podatkov o trenutno izvajajoči besedi, stanju sklada in nekaterih registrih. Poleg vmesnika smo razvili še programsko opremo za zajem podatkov, imenovano zbiralnik. Delovanje opreme smo nato preverili z izvajanjem elementarnih testov.

3.1 Načrtovanje

Načrtovanje smo pričeli z obravnavo problema. To je obsegal predstavitev domene in opisa osnovnih lastnosti. Prvotna rešitev je obsegala implementacijo tolmača z vgrajenim zbiralnikom podatkov, glej sliko 3.1. Njegova

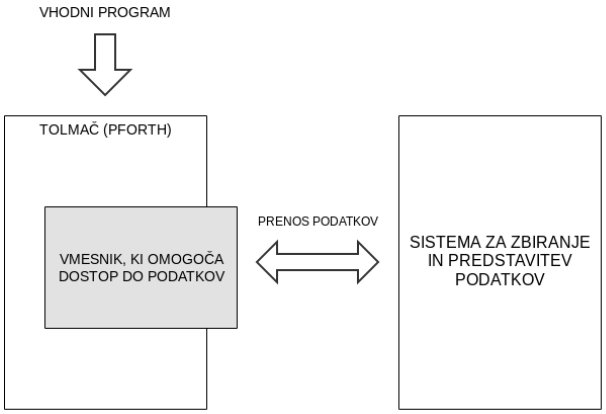
naloga je bila pridobitev, akumulacija in predstavitev podatkov.



Slika 3.1: Začetna rešitve

Po obsežnejšem pregledu domene smo se odločili, da je količina dela prevelika. Zato smo implementacijo tolmača opustili in se osredotočili na nadgradnjo že obstoječe odprto-kodne rešitve. Analizirali smo obstoječe tolmače in izbrali PForth. Z uporabo standardiziranega tolmača smo omogočili izkoristek že obstoječih programov in lažjo nadaljnjo podporo.

Da bi ohranili delovanje tolmača, smo poskusili zmanjšati spremembe na obstoječi kodi. Odrekli smo se uporabi vgrajenega zbiralnika in se osredotočili na vmesnik, ki bo omogočal pridobitev podatkov, ne pa njihove njihove akumulacije in predstavitve. Oblika končne rešitve je prikazana na sliki 3.2.



Slika 3.2: Končna rešitev

3.2 PForth

PForth je program, ki omogoča tolmačenje programskega jezika Forth [4]. Strmi k delovanju znotraj okvirjev ANSI Forth standarda in poudarja prenosljivost, kar je prikazano z začetno črko imena. Črka P predstavlja besedo 'Portable' (slo. *Prenosljiv*), ki predstavlja glavno vodilo implementacije. Jedro PForth tolmača je zapisano v programskem jeziku C.

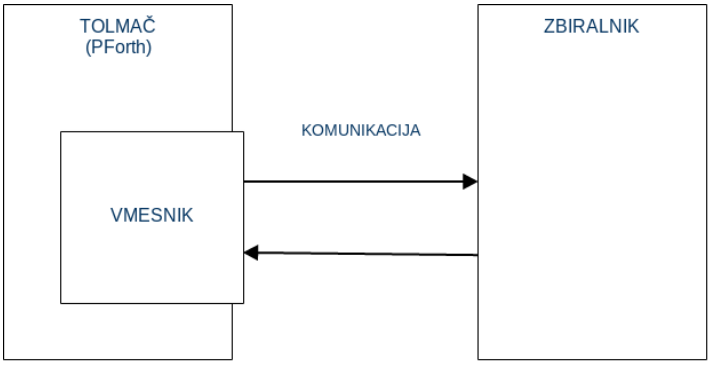
Notranji tolmač se nahaja v funkciji pfCatch, datoteka `pf_forth.c`, in uporablja žetonsko-niteno kodo (ang. *Token-threaded code*). Žetoni predstavljajo zaporedno številko osnovne besede, če je število manjše od števila vseh osnovnih besed, ali pa kažejo na definicijo sestavljene besede izven slovarja. Kot vhodni argument pfCatch funkcije je podan žeton, ki je v funkciji obravnavan s pomočjo *switch* strukture, ki vsebuje obnašanje vseh osnovnih besed. Notranji tolmač je implementiran v jedru ene funkcije, da bi se izognili uporabi prologa in epiloga C-ejevskih funkcij.

3.3 Arhitektura

Rešitev vsebuje dva ločena programa, njuno razmerje pa temelji na odjemalec-strežnik modelu. PForth vmesnik prevzame vlogo strežnika, ki ponuja informacije o stanju tolmača preostalim udeležencem komunikacije. Vlogo od-

jemalca prevzame zbiralnik, ki zbira podatke o stanju in jih predstavi v primerni obliki za nadaljnjo analizo. Strežnik in odjemalec se nahajata na istem sistemu.

Razmerje med posameznimi deli modela je prikazano na sliki 3.3.



Slika 3.3: Odjemalec-strežnik model

3.3.1 Smernice

Naslednje smernice so bile uporabljene pri razvoju nadgradnje:

1. Nadgradnja je prototip in zato optimizacija ni prioriteta.
2. Vpliv nadgradnje na delovanje PForth-a mora biti minimalen.
3. PForth vmesnik ponudi podatke pred začetkom tolmačenja vsakega ukaza.
4. Zgradba vmesnika mora podpirati enostavno dodajanje novih zahtev.
5. Komunikacija mora omogočiti enostaven prenos podatkov in sinhronizacijo.
6. Komunikacija ne sme biti vezana na določen programski jezik.
7. Komunikacija mora biti prilagodljiva.

8. Programski vmesnik mora biti enostaven in razumljiv.

9. Naloga zbiralnika je zbiranje podatkov.

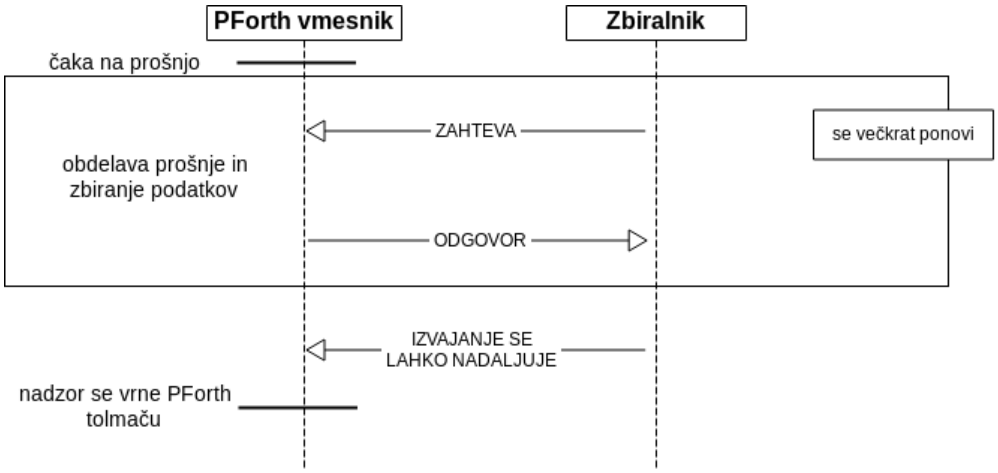
10. Implementacija zbiralnika ni vezana na določen programski jezik.

11. Zbiralnik predstavi podatke na enostaven način.

12. Podatki, ki jih predstavi zbiralnik, so dostopni ostalim programom.

3.4 Komunikacija

Vmesnik in zbiralnik si izmenjujeta podatke preko sporočilne vrste (ang. *Message Queue*) in uporabljata enostaven protokol, ki temelji na Zahteva-odgovor modelu (ang. *Request-Response model*). Za čas komunikacije PForth preda nadzor vmesniku.



Slika 3.4: Komunikacija

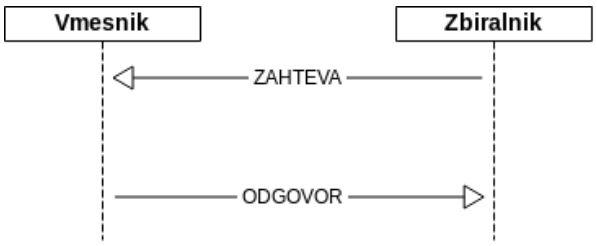
Slika 3.4 prikazuje potek komunikacije med zbiralnikom in vmesnikom. Komunikacija temelji na izmenjavi prošnje in odgovora. Zbiralnik prične izmenjavo s poslano prošnjo, ki jo vmesnik obdela in vrne njene rezultate v

naslednjem odgovoru. Število izmenjav je odvisno od zbiralnika, ki vodi potek komunikacije. Komunikacija se zaključi, ko zbiralnik posreduje sporočilo, ki označuje konec zbiranja podatkov.

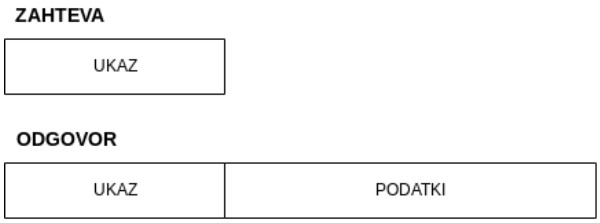
3.4.1 Komunikacijski protokol

Komunikacijski protokol definira obliko, vsebino in uporabo sporočil. Pri njegovi definiciji smo upoštevali smernice 5, 6 in 7, ter s tem zagotovili enostavnost, prenosljivost in prilagodljivo komunikacijo.

Sporočila se delijo na zahteve in odgovore. Zbiralnik kot odjemalec uporablja zahtevo za pridobitev podatkov o stanju tolmača. Vmesnik kot strežnik pa na zahtevo odgovori z odgovorom. Slika 3.5 prikazuje izmenjavo sporočil.



Slika 3.5: Izmenjava sporočil



Slika 3.6: Oblika sporočil

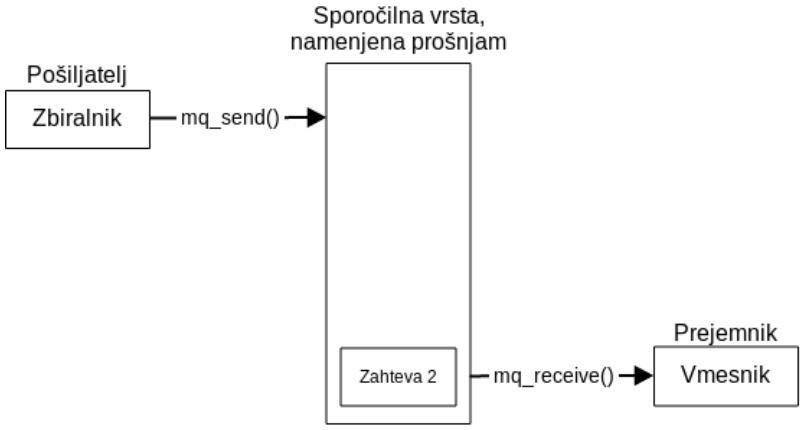
Na sliki 3.6 je prikazana oblika sporočila. Polje UKAZ, ki je del zahteve, predstavlja dejanje, ki ga mora opraviti vmesnik v imenu zbiralnika. Rezultat dejanja je vrnjen v polju PODATKI, katerega oblika je odvisna od dejanja in mora biti znana obema udeležencema komunikacije. Polje UKAZ, ki se

nahaja v odgovoru, pa predstavlja izvedeno dejanje oziroma stanje tolmača. Ponavadi je izvedeno dejanje enako zahtevanemu, le v primeru napake ali zaključka tolmačenja je ta lahko drugačen.

Prenos podatkov bo potekal v serializirani obliki in bo v skladu z MessagePack formatom. Opis formata lahko najdete v poglavju 3.5.

3.4.2 Implementacija komunikacije

Za prenos se uporabljata dve sporočilni vrsti, ki služita kot enosmerni poštni nabiralnik [8]. Prva sporočilna vrsta služi prenosu zahtev od zbiralnika do vmesnika in druga služi prenosu odgovorov v obratni smeri. Slika 3.7 prikazuje izmenjavo zahteve.

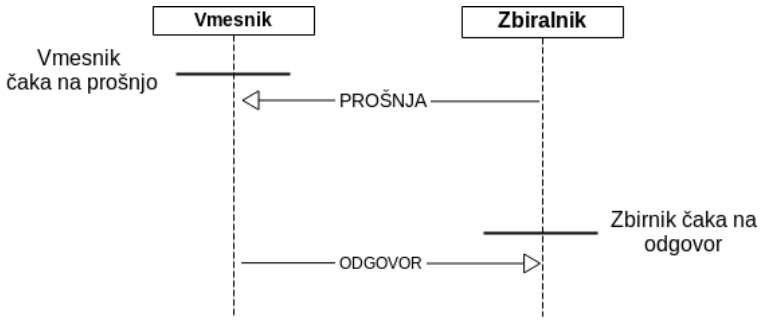


Slika 3.7: Sporočilna vrsta

V GNU/Linux operacijskem sistemu se sporočilna vrsta nahaja v delu polnilnika, ki pripada jedru operacijskega sistema. Za upravljanje s sporočilno vrsto je odgovoren operacijski sistem, ki ponuja programski vmesnik za uporabo. Proces se lahko pridruži ali ustvari sporočilno vrsto z uporabo funkcije mq_open(), v kateri kot enega izmed argumentov podamo ime sporočilne vrste. Dva procesa lahko uporabljata isto sporočilno vrsto, če podata isto ime. Za dodajane sporočil uporabimo klic mq_send() in za prevzem sporočila mq_receive(). Sporočila se obravnavajo kot zaključena enota, kar pomeni, da

uporabniku ni treba skrbeti za integriteto sporočila. Ko proces konča s komunikacijo, se odreče uporabi sporočilne vrste s klicem funkcije mq_close(). Proces lahko zahteva uničenje sporočilne vrste s klicem mq_unlink().

Proces z uporabo funkcije mq_receive() prevzame sporočilo. Če je sporočilna vrsta prazna, se proces prestavi v ozadje in prepusti procesorski čas preostalim uporabnikom. Proces bo nadaljeval svoje izvajanje, ko vrsta prejme novo sporočilo. To omogoči med-procesno sinhronizacijo, ki je potrebna zaradi odvisnosti procesa od vsebine sporočil. Slika 3.8 prikazuje mesta sinhronizacije.



Slika 3.8: Sinhronizacija

Vmesnik in zbiralnik vzpostavita komunikacijo ob začetku delovanja s klicem funkcije mnt_com_srv_start() in mnt_com_cli_start(). Udeleženca kličeta funkciji vzajemno z njuno vlogo v arhitekturi.

Zapis 3.1 prikazuje funkcijo mnt_com_src_start() iz datoteke mnt_common.c.

```
1  bool mnt_com_srv_start(mqd_t *mq, const char *name)
2  {
3      struct mq_attr attr;
4
5      /* try to unlink the message queue */
6      mq_unlink(name);
7
8      attr.mq_flags = 0;
9      attr.mq_maxmsg = 10;
10     attr.mq_msgsize = MNT_MQ_MAX_MSG_LENGTH;
11     attr.mq_curmsgs = 0;
12
```

```
13      *mq = mq_open(name, O_CREAT | O_RDWR, 0644, &attr);
14      if((int)(*mq) == -1)
15      {
16          perror("Error: mq_open");
17          return false;
18      }
19
20      return true;
21  }
```

Primer 3.1: Funkcija mnt_com_src_start()

Po uspešni vzpostavitvi komunikacije lahko pričnemo s pošiljanjem sporočil. Preden je sporočilo poslano, je treba podatke pretvoriti v MessagePack format. Zato uporabljamo CMP knjižnico [4], ki omogoča enostavno pretvorbo in je zapisana v programskem jeziku C, kompatibilnim z ANSI C Standardom.

Funkcija mnt_build_request(), primer 3.2, ki se nahaja v datoteki mnt_common.c, je primer funkcije, namenjene gradnji sporočila.

```
/**
 * @brief Build a request message.
 * @param buffer A buffer used for the request message storage.
 *      Has to be large enough.
 * @param cmd A command used in the message.
 * @param Returns false if message transformation was not
   successful.
 */
bool mnt_build_request(char *buffer, const uint16_t cmd)
{
    cmp_ctx_t cmp;
    cmp_init(&cmp, (void *)buffer,
             cmp_buffer_reader,
             cmp_buffer_writer);
    cmp_writer_pointer_reset();

    if(!cmp_write_array(&cmp, 1))
    {
        fprintf(stderr, "Error: mnt_build_cmd:"
          "cmp_write_array!\n");
        return false;
    }

    if(!cmp_write_u16(&cmp, cmd))
    {
        fprintf(stderr, "Error: mnt_build_cmd:"
          "cmp_write_u16!\n");
        return false;
    }

    return true;
}
```

Primer 3.2: Funkcija mnt_build_request()

3.5 Serializacija in deserializacija podatkov

Serializacija (ang. *Serialization*) je proces pretvorbe podatkovne strukture v obliko, ki je primerna za hranjenje in pošiljanje podatkov med različnimi sistemi ali programskimi jeziki [9]. Rezultat serializacije je niz bitov, ki omogoča rekonstrukcijo semantično identične podatkovne strukture. Nosilci informacije o strukturi so metapodatki, ki so dodani med serializacijo. Proces rekonstrukcije se imenuje deserializacija (ang. *Deserialization*).

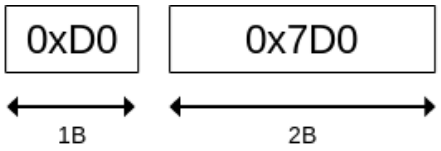
MessagePack je podatkovni format, ki omogoča predstavitev preprostih podatkovnih struktur v serializirani binarni obliki [3]. Zasnova formata strmi k majhnosti in kompaktnosti. Programska podpora je na voljo v več programskih jezikih, kar omogoča visoko prenosljivost. MessagePack za opis podatkovnih struktur uporabi kombinacijo tipa in formata. Sistem tipov omogoča razvrstitev podatkov glede na njihovo uporabo in množico mogočih vrednosti. Sistem vsebuje pogoste osnovne tipe, kot so Integer, Floar, String in Map, ki so nadaljnjo razvrščeni z uporabo formata. Format predstavi dodatne omejitve množice in določa zaporedje zapisanih bitov.

Format int16 ima šestnajstiško predstavitev 0xD0 in je tipa Integer, katerega množica mogočih vrednosti je $(-2^{63}, 2^{63}-1)$. Dodatna omejitev formata zmanjša množico vrednosti na $(-2^{15}, 2^{15}-1)$ in predpostavi, da je zaporedje zapisanih bitov v *big-endian* obliki. Omejitev množice mogočih vrednosti je prikazana v imenu formata s številom 16, ki predstavlja dolžino bitov, s katero je vrednost lahko zapisana [3].

Primer zapisa spremenljivke 'stevilo', primer 3.3, v MessagePack formatu je prikazan na sliki 3.9.

```
int16_t stevilo = 0x7D0;
```

Primer 3.3: Spremenljivka 'stevilo' zapisana v C programskem jeziku.

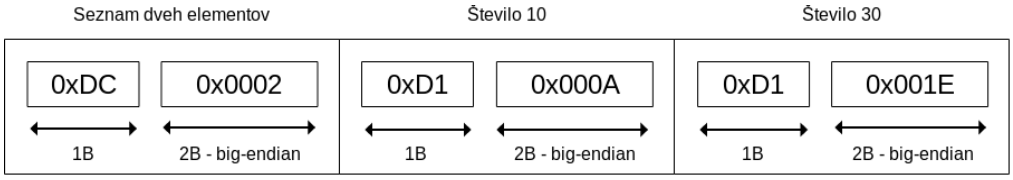


Slika 3.9: Oblika spremenljivke 'stevilo' v MessagePack formatu

Vrednost 0xD0 je metapodatek, katerega pomen je format int16, kar pomeni, da naslednji podatek predstavlja celo število z množico mogočih vrednosti $(-2^{15}, 2^{15} - 1)$. Zapis kompleksnejše strukture je prikazan na sliki 3.10.

```
int16_t seznam_x[] = {10, 30};
```

Primer 3.4: Seznam 'seznam_x' zapisana v C programskem jeziku



Slika 3.10: Seznam 'seznam_x', zapisana v MessagePack formatu, primer 3.4.

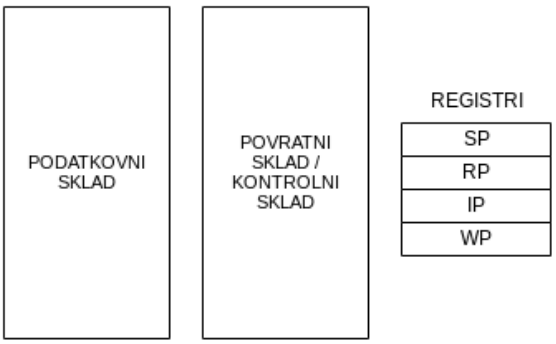
Vrednost 0xDC predstavlja format arr16, ki napoveduje seznam N elementov. N je vrednost, zapisana v naslednjih šestnajstih bitih in je v tem primeru 0x2. Tako izvemo, da naslednji podatki predstavljajo seznam dveh vrednosti.

3.6 Vmesnik

PForth tolmač je navidezni stroj, ki je namenjen tolmačenju programskega jezika Forth. Njegovi deli omogočajo 'izvajanje' programa in so vir informacij o njegovemu delovanju. Če želimo zbrati informacije o delovanju programa, moramo torej zbrati informacije o samem stroju. Namen PForth vmesnika je oskrbeti svet s temi informacijami.

3.6.1 Zajem podatkov

Delovanje navideznega stroja lahko predstavimo kot set zaporednih stanj, kjer stanje predstavlja vsebina vseh pomembnih delov sistema v določenem trenutku. V poglavju 2.6 smo predstavili osnovne elemente navideznega stroja, brez katerih ta ne more delovati. Poiskali smo njihove predstavnike v tolmaču in obravnavali njihovo delovanje. Predpostavili smo, da so podatkovni sklad, povratni sklad, kontrolni sklad in vsi registri primerna predstavitev sistema, glej sliko 3.11.



Slika 3.11: Stanje sistema

3.6.2 Interval zajemanja

Interval zajemanja je direktno povezan z količino zajetih podatkov. Če podatke zajemamo s premajhno periodo, pride do fragmentacije informacij. Če

je perioda zajemanja prevelika, lahko izpustimo stanje, ki nosi pomembno informacijo.

Smiselno je, da omejimo izbiro periode na mnogokratnike časa ene izvedene besed, saj uporaba periode manjše od časa ene izvedene besede povzroči ponavljanja IP in WP registrov, ki kažeta na trenutno izvajajočo in naslednjo besedo. Ponavljanje registrov pripomore k fragmentaciji informacij čemur se želimo izogniti.

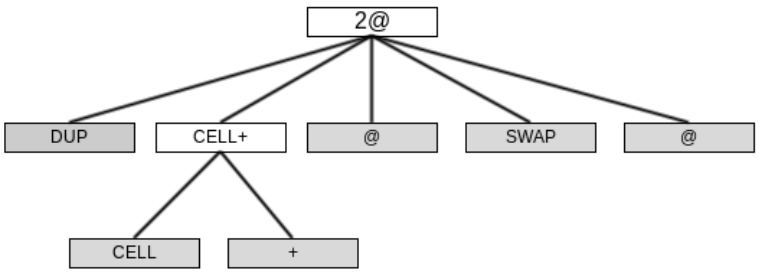
Forth programski jezik se močno zanaša na uporabo sestavljenih besed. Iz poglavja 2.3 vemo, da so sestavljene besede zgrajene iz več osnovnih in sestavljenih besed, ki se obravnavajo ob tolmačenju. Beseda 2@ ima sledečo definicijo

```
: 2@ ( addr -- x1 x2 )
    DUP CELL+ @ SWAP @
;
```

in je sestavljena iz ene sestavljene in štirih osnovnih besed. CELL+ je sestavljena beseda, ki ima sledečo definicijo in je zgrajena iz dveh osnovnih besed.

```
: CELL+ ( n -- n+cell )
    CELL +
;
```

Slika 3.12 prikazuje drevesno zgradbo besede 2@.



Slika 3.12: Drevesna zgradba beseda 2@

Skupno je beseda 2@ sestavljena iz šestih osnovnih besed. Predpostavimo,

da nas zanima stanje pred in po končani funkciji 2@ in da je perioda zbiranja podatkov enaka eni izvedeni osnovni besedi. Med tolmačenjem besede 2@ bomo izvedli šest osnovnih besed in posledično bomo zbrali tudi šest stanj. Torej šest stanj predstavi tolmačenje besede 2@. Težava se pojavi, ker nas zanima samo stanje pred in po tolmačenju besede 2@, kar pomeni, da nas vmesna stanja ne zanimajo. Namesto dveh zajemov jih naredimo šest. Posledično imamo večjo količino podatkov. Prepogost zajem podatkov prispeva k fragmentaciji informacij in oteži analizo. Težava se poveča z uporabo kompleksnejših besed.

Nasprotje zgornjemu primeru je zajem podatkov pred in po tolmačenju sestavljene besede. Na primeru besede INFINITY_LOOP, ki vsebuje neskončno zanko, lahko vidimo, da preredek zajem podatkov lahko povzroči izgubo informacij. Ker podatke zajemamo samo pred in po sestavljeni besedi, zbirnik ne bo prejel ponavljajočih se stanj neskončne zanke in posledično bo analiza programa napačna.

```
: INFINITY_LOOP
    BEGIN
          I . CR
           TRUE
    AGAIN
;
```

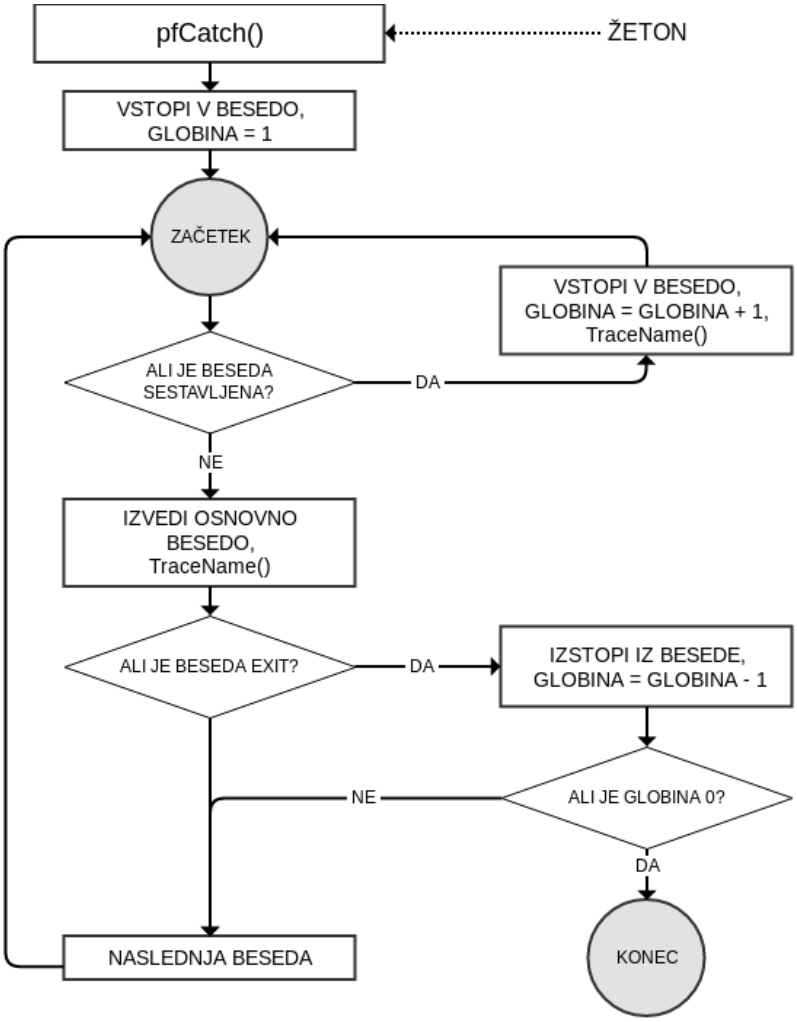
Razvidno je, da mora biti interval zajema podatkov primerno dolg. Premajhen interval povzroči fragmentacijo informacij, predolg interval pa lahko povzroči izgubo podatkov. Rešitev, ki smo jo uporabili, je predstavljena v poglavju 3.6.3 in je vezana na globino tolmačene besede.

3.6.3 Zbirna točka

Dostop do gradnikov tolmača, ki predstavljajo stanje sistema, ni enak skozi celoten tolmač. Dostop do gradnika je omejen z življenjsko dobo objekta in območja njegove veljavnosti. Z dekonstrukcijo tolmača smo poiskali točko, ki ima primeren dostop do vseh gradnikov.

Funkcija pfCatch() vsebuje delovanje notranjega tolmača. Znotraj so tolmačene sestavljene in se izvajajo osnovne besede. Funkcija ima dostop do vseh osnovnih delov in je hkrati edino mesto, kjer lahko spremljamo izvajanje osnovnih besed. Pomembno je, da se zbirna točka nahaja znotraj te funkcije.

Delovanje funkcije pfCatch() prikazuje diagram 3.13.



Slika 3.13: Delovanje funkcije pfCatch()

Spremenljivka GLOBINA sledi vstopom in izstopom tolmača v sestavljene besede. Ko je globina enaka nič, se tolmačenje zaključi. Funkcija

TraceNames() omogoča uporabniku sledenje besedam. Na podlagi žetona izpiše ime izvedene ali tolmačene besede. Ta lastnost je posebnost PForth tolmača in ni del ANSI Forth standarda.

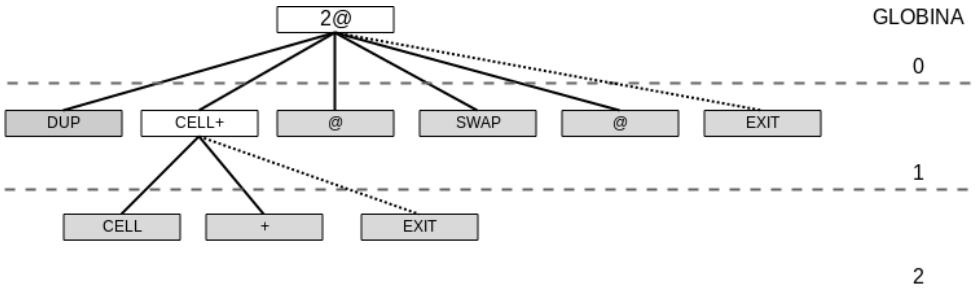
Izpis 3.5 prikazuje tolmačenje besede 2@.

```
>2@
>   DUP
>   CELL+
>      CELL
>      +
>      EXIT
>   @
>   SWAP
>   @
>   EXIT
```

Primer 3.5: Izpis tolmačene besede 2@

Odmik besede, od leve proti desni, prikazuje vstop v sestavljeno besedo. Torej so besede, ki jim sledijo levo zamaknjene besede, sestavljene (2@, CELL+). Preostale besede so lahko sestavljene ali osnovne. Vstop ali izstop iz sestavljene besede poveča ali zmanjša globino.

Slika 3.14 prikazuje izpis v obliki drevesa.



Slika 3.14: Globina besed

Globino sledenja oziroma izpisa lahko nadziramo z zapisom največje globine v spremenljivko TRACE-LEVEL. Spremenljivko lahko nastavimo z naslednjim ukazom

```
<globina> TRACE-LEVEL !
```

kjer oznako <globina> nadomestimo z največjo globino. Primer 3.6 prikazuje sledenje besedi 2@ z največjo globino tri, primer 3.7 prikazuje sledenje z največjo globino dva in primer 3.8 z ena.

```
3 TRACE-LEVEL !
2@
>2@
>   DUP
>   CELL+
>      CELL
>      +
>      EXIT
>   @
>   SWAP
>   @
>   EXIT
```

Primer 3.6: Sledenje besede 2@ z največjo globino tri

```
2 TRACE-LEVEL !
2@
>2@
>   DUP
>   CELL+
>   @
>   SWAP
>   @
>   EXIT
```

Primer 3.7: Sledenje besede 2@ z največjo globino dve

```
1 TRACE-LEVEL !
2@
>2@
```

Primer 3.8: Sledenje besede 2@ spremljanja z največjo globino ena

Mehanizem sledenja lahko služi kot primeren nadzor intervala zajemanja podatkov. Posledica uporabe je odvisnost periode od globine izvedene

besede, kar pomeni, da perioda zajema ni več konstantna.

Z definicijo vmesnika znotraj funkcije TraceNames(), ki je večkrat uporabljena znotraj funkcije pfCatch(), slika 3.13, zadostimo potrebama dostopnosti gradnikov in nadzoru periode zajema.

3.6.4 Programski vmesnik

Programski vmesnik je definiran v funkciji TraceNames(), pogavlje 3.6.3, in podpira komunikacijo, opisano v poglavju 3.4. Delovanje je sestavljeno iz:

- prejema zahteve,
- razrešitve zahteve in
- odgovora na zahtevo.

Zahteva je razrešena na podlagi prejetega ukaza. Ukaz je obdelan z uporabo *switch* strukture, ki za vsak ukaz izvede določen del kode. Izveden del kode vsebuje izbiro podatkov in njihovo pripravo na prenos. Zahteva in odgovor sta v skladu s poglavjem 3.4.

Ukazi za zbiranje podatkov:

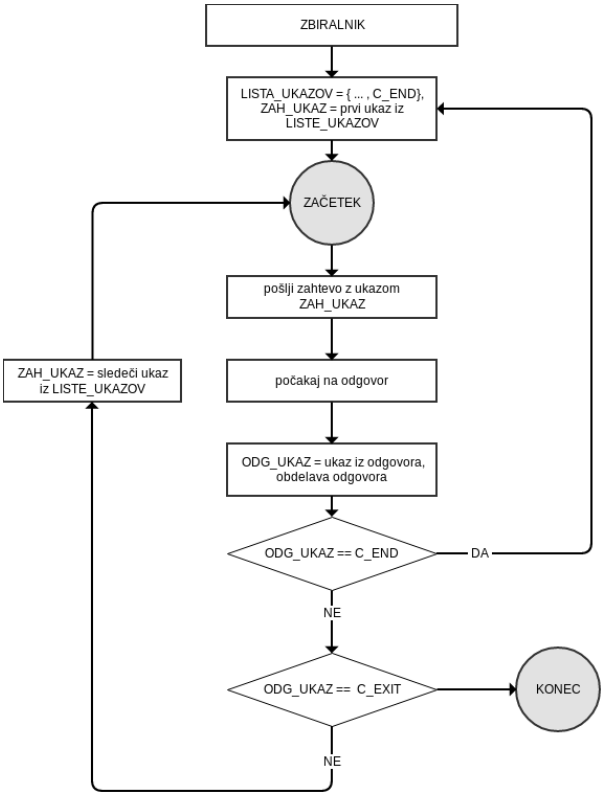
Ukaz	Odgovor
C_REG_SP	Odgovor vsebuje SP register.
C_REG_RP	Odgovor vsebuje RP register.
C_DATA_STATE	Odgovor vsebuje Podatkovni in Povratni/Kontrolni sklad ter ime izvedene besede.

Ukazi za sinhronizacijo programa:

Ukaz	Pomen
C_END	Tolmač lahko nadaljuje s tolmačenjem besede.
C_EXIT	Označuje konec tolmačenja programa.

3.7 Zbiralnik

Zbiralnik je program, ki preko vmesnika dostopi do podatkov in jih izpiše na standardni izhod v primerni obliki za nadaljnjo analizo. Za dostop do podatkov uporablja vmesnik definiran v poglavju 3.6.4. Izmenjava sporočil je v skladu s poglavjem 3.4.



Slika 3.15: Delovanje zbiralnika

Zbiralnik je definiran v datoteki monitor.c. Njegovo delovanje je opisano v diagramu 3.15. Komunikacija se prične s poslano zahtevo. Sledi čakanje na odgovor in korak obravnave. Obravnava je odvisna od vrnjenega ukaza. Pri obravnavi prejetega ukaza uporabimo *switch* strukturo, ki pravilno deserializira podatke in jih izpiše na standardni izhod. Primer 3.9 prikazuje obravnavo odgovora.

```
while(...)
{
  /* pošlji zahtevo  */
  mnt_build_request(..., cmd_sequence[cmd_sequence_cnt]);
  mq_send(...);

  /* čakanje na odgovor */
  mq_receive();

  /* obdelava odgovora */
  mnt_parse_cmd(..., &cmd))
  switch(cmd)
  {
  case C_DATA_STATE
    // obravnava C_DATA_STATE sporočila
    break;
  case C_REG_SP:
  case C_REG_RP:
    // obravnava C_DATA_STATE sporočila
    break;
  case C_EXIT:
    // zapusti zanko
    break;
  default:
    // neznan ukaz
    return EXIT_FAILURE;
  }
  cmd_sequence_cnt++;
}
```

Primer 3.9: Simbolični prikaz delovanja zbirnika v C programskem jeziku

Ob končanem tolmačenju vmesnik vrne ukaz C_EXIT, ki povzroči izhod iz zanke in konec zbiranja podatkov. Primer 3.10 prikazuje primer izpisa prejetih podatkov.

```
19391920:19396072:I::,0,70078431232,0
19391912:19396072:.:,0:,0,70078431232,0
19391920:19396072:CR::,0,70078431232,0
```

```
4 19391920:19396072:(LOOP)::,0,70078431232,0
```

Primer 3.10: Primer izpisa

Zaporedje poslanih zahtev je C_REG_SP, C_RE_RP, C_DATA_STATE, in C_END. Slika 3.16 prikazuje interpretacijo izpisa.

Slika 3.16: Predstavitev izpisa

Poglavje 4

Uporaba

Naslednje poglavje predstavi gradnjo in uporabo programske opreme ter opisuje izvedene teste, ki so potrdili delovanje nadgradnje.

Koda programske opreme je dostopna preko `https://github.com/siks89/pfextend` povezave in je namenjena uporabi na GNU/Linux operacijskem sistemu.

4.1 Priprava programov

Programsko kodo lahko prenesemo na osebni računalnik preko sledeče povezave `https://github.com/siks89/pfextend` z uporabo git kontrolnega sistema.

```
> git clone https://github.com/siks89/pfextend
```

Gradnjo programa smo poenostavili z uporabo GNU Make programske opreme. Navodila za gradnjo se nahajajo v Makefile datoteki. Programsko opremo lahko zgradimo s klicem ukaza 'make'.

```
> cd pfextend
> make
```

Po uspešni gradnji se v datoteki pfextend pojavi datoteka out, ki vsebuje naslednje datoteke.

Program	Opis
pforth_standalone	Nadgrajeni PForth tolmač
monitor.out	Zbiralnik
analyser.out	Program za analizo pridobljenih podatkov
auto_run_monitor.sh	Skripta za enostaven zagon tolmača in Zbiralnika
auto_run_analyser.sh	Skripta za enostaven zagon tolmača, Zbiralnika in testnega programa za analizo (analyser.out)

4.2 Uporaba programa

Po uspešni gradnji programov, glej poglavje 4.1, lahko pričnemo z zbiranjem podatkov. Sprva poženemo PForth tolmač. Uporabimo ukaz

```
> ./pforth_standalone -m <pot_do_programa>
```

Zastavica "m" omogoči delovanje vmesnika in parameter <pot_do_programa> kaže na program, ki ga hočemo izvesti. Nato zaženemo zbiralnik v ločenem terminalu. Uporabimo ukaz

```
> ./monitor
```

Tolmačen program z nastavitvijo globine označuje začetek zbiranja podatkov.

```
<globina> TRACE_LEVEL !
```

Simbol <globina> predstavlja največjo globino sledenih besed, glej poglavje 3.6.3.

Sledeči program je primer končne zanke.

```
\ Definicija FINITE_LOOP zanke.
: FINITE_LOOP
    1000 1
    DO
    LOOP
;
```

```
\ Izpis "EXE: FINITE_LOOP"
.( EXE: FINITE_LOOP) CR

\ Nastavimo globino sledenja.
2 TRACE-LEVEL !

\ Zbiranje podatkov je omogočeno.

\ Izvedba zanke.
FINITE_LOOP
```

Ko ga zaženemo, lahko pričakujemo naslednji izpis

```
Monitor:
12050864:12055040:FINITE_LOOP::
12050864:12055032:(LITERAL)::,0
12050856:12055032:(LITERAL):,1000:,0
12050848:12055032:(DO):,1,1000:,0
12050864:12055016:(LOOP)::,1000,1,0
12050864:12055016:(LOOP)::,1000,2,0
...
12050864:12055016:(LOOP)::,1000,996,0
12050864:12055016:(LOOP)::,1000,997,0
12050864:12055016:(LOOP)::,1000,998,0
12050864:12055016:(LOOP)::,1000,999,0
12050864:12055032:EXIT::,0
12050864:12055040:AUTO.TERM::
12050864:12055032:HISTORY.OFF::,0
12050864:12055032:AUTO.TERM::,0
12050864:12055032:EXIT::,0
```

Oznaka "..." označuje ponavljanje izpisa.

Za enostaven zagon tolmača in zbiralnika lahko uporabimo ukaz

```
./auto_run_monitor.sh <pot_do_programa>
```

Zbrane podatke lahko preusmerimo v program, namenjen analizi podatkov. Uporabimo

```
> ./auto_run_monitor.sh <pot_do_programa> |
  <program_za_analizo_podatkov>
```

kjer simbol <program_za_analzo_podatkov> predstavlja program, ki preko standardnega vhoda prevzame podatke in jih analizira.

4.3 Testi

Delovanje nadgradnje smo potrdili z elementarnimi testi. Ustvarili smo program 'analyser.out', ki na podlagi izbranih podatkov napove ustavljivost programa. Program med zbranimi podatkih išče zaporedje ponavljajočih se vrstic, kjer vsaka vrstica predstavlja stanje tolmača. Uporabimo ga lahko z ukazom

```
./auto_run_monitor.sh <pot_do_programa> | analyser.out
```

ali pa uporabimo skripto

```
./auto_run_analyser.sh <pot_do_programa>
```

za enostavnejšo uporabo. Delovanja programa lahko prikažemo na primeru

```
printf "14\n1\n20\n16\n20\n16\n20\n16\n20\n16\n20\n16" |
    ./analyser.out
```

kjer ukaz

```
printf "14\n1\n20\n16\n20\n16\n20\n16\n20\n16\n20\n16"
```

predstavlja sledeče vhodne podatke

```
14
1
20
16
20
16
20
16
20
16
20
16
```

Program zazna ponavljanje števila 20 in 16. Program temelji na domnevi, da lahko iz trenutnega stanja sistema napovemo stanje, ki sledi. Torej, če je simbolično stanje sistema 20 in temu sledi enako stanje 20, lahko sklepamo, da stanje 20 povzroči stanje 20. Napovemo lahko, da se stanje sistema ne bo spremenilo. Enako velja za zaporedno vrsto stanj. Če je trenutno stanje sistema

```
20
16
```

povzroči zaporedje

```
20
16
```

lahko sklepamo, da se ponavljanje zaporedja ne bo končalo. Domneva se zanaša na deterministično obnašanje in celovitost predstavitve stanja sistema. V primeru, ko to ne drži, domneva ni veljavna.

Testni primer 4.1 predstavlja neskončno zanko.

```
: INFINITY_LOOP
    BEGIN
    AGAIN
;

.( EXE: INFINITY_LOOP ) CR

2 TRACE-LEVEL !

INFINITY_LOOP
```

Primer 4.1: Prime neskončne zanke

Uporabimo ukaz

```
> ./auto_run_analyser.sh ../example/infinite_loop.f
```

in dobimo naslednji izpis

```
1 RUN: ../example/infinite_loop.f
2 Analyser:
3 Line: Monitor:
```

```
Line: 35033520:35037696:INFINITY_LOOP::
Line: 35033520:35037688:BRANCH::,0
Line: 35033520:35037688:BRANCH::,0
Prediction: Program will not STOP!
```

Razvidno je, da program analyser.out lahko prepozna enostavno ponavljanje stanj in z upoštevanjem deterministočnosti sistema napove, da se program ne bo ustavil. Primer 4.2 predstavlja končno zanko s tisoč ponovitvami.

```
: FINITE_LOOP
    1000 1
    DO
    LOOP
;

.( EXE: FINITE_LOOP) CR

2 TRACE_LEVEL !

FINITE_LOOP
```

Primer 4.2: Primer FINITE_LOOP

Z analizo dobimo naslednji izpis

```
RUN: ../example/finite_loop.f
Monitor:
12050864:12055040:FINITE_LOOP::
12050864:12055032:(LITERAL)::,0
12050856:12055032:(LITERAL):,1000:,0
Line: 18977184:18981368:(DO):,1,1000:,0
Line: 18977200:18981352:(LOOP)::,1000,1,0
Line: 18977200:18981352:(LOOP)::,1000,2,0
Line: 18977200:18981352:(LOOP)::,1000,3,0
Line: 18977200:18981352:(LOOP)::,1000,4,0
Line: 18977200:18981352:(LOOP)::,1000,5,0
Line: 18977200:18981352:(LOOP)::,1000,6,0
Line: 18977200:18981352:(LOOP)::,1000,7,0
...
Line: 12050864:12055016:(LOOP)::,1000,996,0
```

```
16 Line: 12050864:12055016:(LOOP)::,1000,997,0
17 Line: 12050864:12055016:(LOOP)::,1000,998,0
18 Line: 12050864:12055016:(LOOP)::,1000,999,0
19 Line: 12050864:12055032:EXIT::,0
20 Line: 12050864:12055040:AUTO.TERM::
21 Line: 12050864:12055032:HISTORY.OFF::,0
22 Line: 12050864:12055032:AUTO.TERM::,0
23 Line: 12050864:12055032:EXIT::,0
24 Program has stopped!
```

Oznaka "..." označuje ponavljanje izpisa. Analiza primera ne napove neskončnega izvajanja, saj izpis ne vsebuje ponavljajočih se stanj. Iz Povratnega/Kontrolnega sklada je razvidna, slika 4.1, iteracija števca (0, 1, 2, ... , 999) in ohranjanje primerjalne vrednosti 1000.

Slika 4.1: Primer stanja

Program 4.3 je primer neskončne zanke, ki izpisuje lihost ali sodost trenutnega koraka. Jedro zanke vsebuje dodatno kodo, ki ne vpliva na izvajanje zanke, in s tem predstavi dodatno kompleksnost, ki uteži analizo.

```
: INFINITY_LOOP_PRINT
    0
    BEGIN
        ." To je neskoncna zanka." CR
        0=
        IF
            ." Lihi korak!" CR
            1
        ELSE
            ." Sodi korak!" CR
            0
```

```
        THEN
    AGAIN
;

.( EXE: INFINITY_LOOP_PRINT ) CR

2 TRACE-LEVEL !

INFINITY_LOOP_PRINT
```

Primer 4.3: Primer INFINITY_LOOP_PRINT

Rezultat analize

```
RUN: ../example/infinite_loop_print.f
Analyser:
Line: Monitor:
Line: 31007152:31011328:INFINITY_LOOP_PRINT::
Line: 31007152:31011320:(LITERAL)::,0
Line: 31007144:31011320:(.")::,0:,0
Line: 31007144:31011320:CR:,0:,0
Line: 31007144:31011320:0=:,0:,0
Line: 31007144:31011320:0BRANCH:,18446744073709551615:,0
Line: 31007152:31011320:(.")::,0
Line: 31007152:31011320:CR::,0
Line: 31007152:31011320:(LITERAL)::,0
Line: 31007144:31011320:BRANCH:,1:,0
Line: 31007144:31011320:BRANCH:,1:,0
Line: 31007144:31011320:(.")::,1:,0
Line: 31007144:31011320:CR:,1:,0
Line: 31007144:31011320:0=:,1:,0
Line: 31007144:31011320:0BRANCH:,0:,0
Line: 31007152:31011320:(.")::,0
Line: 31007152:31011320:CR::,0
Line: 31007152:31011320:(LITERAL)::,0
Line: 31007144:31011320:BRANCH:,0:,0
Line: 31007144:31011320:(.")::,0:,0
Line: 31007144:31011320:CR:,0:,0
Line: 31007144:31011320:0=:,0:,0
```

```
Line: 31007144:31011320:0BRANCH:,18446744073709551615:,0
Line: 31007152:31011320:(."):,0
Line: 31007152:31011320:CR:,0
Line: 31007152:31011320:(LITERAL):,0
Line: 31007144:31011320:BRANCH:,1:,0
Line: 31007144:31011320:BRANCH:,1:,0
Line: 31007144:31011320:(."):,1:,0
Line: 31007144:31011320:CR:,1:,0
Line: 31007144:31011320:0=:,1:,0
Line: 31007144:31011320:0BRANCH:,0:,0
Line: 31007152:31011320:(."):,0
Line: 31007152:31011320:CR:,0
Line: 31007152:31011320:(LITERAL):,0
Line: 31007144:31011320:BRANCH:,0:,0
Line: 31007144:31011320:(."):,0:,0
Prediction: Program will not STOP!
```

Kljub dodatni kodi, ki jo program izvede ob vsaki ponovitvi zanke, je program analyser.out uspel prepoznati, da se program ne bo ustavil.

Predhodnih primeri so potrdili delovanje nadgradnje. Pokazali smo, da je mogoče v preprostih primerih ločiti med končno in neskončno zanko. Čeprav so primeri enostavni, menimo, da lahko s prilagajanjem nadgradnje omogočimo analizo kompleksnejših programov.

Poglavje 5

Sklepna ugotovitev

Namen diplomskega dela je bila izdelava programske opreme, ki podpira opazovanje in analizo konkatenacijskega jezika. Uporaba smernic nas je vodila k izbiri programskega jezika Forth. Preko načrtovanja in analize smo postavili osnovno arhitekturo in se odločili za uporabo PForth tolmača. Preučili smo delovanje navideznega stroja in s tem določili opis stanja sistema. V tolmaču smo poiskali izvorne točke podatkov in temu prilagodili arhitekturo nadgradnje. Definirali smo način komunikacije in izdelali nadgradnjo z vso podporno opremo. Testi so pokazali uspešnost in primernost nadgradnje za nadaljnje poizkuse. Izvedeni testi so bili preprosti, zato menimo, da bo z dvigom kompleksnosti nastala potreba po razširitvi nadgradnje in izboljšavi metode analize, kar je bilo predvideno skozi proces izdelave. Nadgradnja je zadostna in primerna za uporabo. Nadaljnja smer dela vključuje izboljšavo opreme in metode analize.

Literatura

[1] Forth language processor on comet.

http://octodecillion.com/blog/forth-language-processor-on-comet/, 2014. [Online; obiskan 10-Maj-2017].

[2] FreeBSD man pages - loader. `https://www.freebsd.org/cgi/man.cgi?loader(8)`, 2015. [Online; obiskan 10-Maj-2017].

[3] Messagepack specifications. `https://github.com/msgpack/msgpack/blob/master/spec.md`, 2015. [Online; obiskan 10-Maj-2017].

[4] Cmp library. `https://github.com/camgunz/cmp`, 2016. [Online; obiskan 10-Maj-2017].

[5] Concatenative language. `https://concatenative.org/wiki/view/Concatenative%20language`, 2016. [Online; obiskan 10-Maj-2017].

[6] Concatenative programming language. `https://en.wikipedia.org/wiki/Concatenative_programming_language`, 2016. [Online; obiskan 10-Maj-2017].

[7] Forth (programming language). `https://en.wikipedia.org/wiki/Forth_(programming_language)`, 2016. [Online; obiskan 10-Maj-2017].

[8] Message queue. `http://man7.org/linux/man-pages/man7/mq_overview.7.html`, 2016. [Online; obiskan 10-Maj-2017].

[9] Serialization. `https://en.wikipedia.org/wiki/Serialization`, 2016. [Online; obiskan 10-Maj-2017].

[10] James R Bell. Threaded code. *Communications of the ACM*, 16(6), 1973.

[11] Elizabeth D. Rather Edward K. Conklin. *Forth Programmer's handbook*, chapter 1, pages 17–46. BookSurge Publishing, 3 edition, 2010.

[12] Elizabeth D. Rather Edward K. Conklin. *Forth Programmer's handbook*, chapter 1, pages 25–28. BookSurge Publishing, 3 edition, 2010.

[13] Elizabeth D. Rather Edward K. Conklin. *Forth Programmer's handbook*, chapter 1, pages 31–33. BookSurge Publishing, 3 edition, 2010.

[14] Brian Hayes. My life as a forth interpreter. 1986.

[15] American National Standards Institute. Forth ANSI standard. `http://www.forth.org/svfig/Win32Forth/DPANS94.txt`, 1994. [Online; obiskan 10-Maj-2017].

[16] R. G. Loeliger. *Threaded Interpretive Languages*, chapter 1, pages 1–8. BYTE Publications, Inc., 1 edition, 1981.

[17] R. G. Loeliger. *Threaded Interpretive Languages*, chapter 2, pages 9–28. BYTE Publications, Inc., 1 edition, 1981.

[18] Brad Rodriguez. Concatenative language. `http://www.bradrodriguez.com/papers/moving1.htm`, 1993. [Online; obiskan 10-Maj-2017].